FPGA Console 设计报告
——硬件实现的 VT220 兼容终端

计 63　陈晟祺　2016010981
计 64　周聿浩　2016011347

目录

1 项目概述

1.1 项目背景

Console，中文名为控制台，是一种计算机的物理设备。它只具有基本的 I/O 功能，而不具有计算能力。在计算机刚诞生时，控制台是与计算机独立的。它一般具有键盘与屏幕，将用户输入传送给计算机，将计算机的输出回显给用户。

随着计算机的发展，越来越少的计算机采取这种分离的设计，操作系统都会内置虚拟控制台（Virtual Console），通过软件即可模拟出控制台的功能。实现这类技术的软件被称为终端模拟器（Terminal Emulator），如 Linux 下的 `xterm`, `gnome-terminal`, `Konsole`, `screen`，Windows 下的 `cmd.exe`、超级终端等工具都属于终端模拟器。另外，借助于称为“虚拟终端”（Virtual Terminal）的技术，也不再有一台计算机只能由一个用户使用的限制。

1.2 功能与特性

本项目实现的是一个真正的**物理**终端！它可以通过串口与计算机连接，通过 PS/2 接口连接键盘，HDMI 接口连接显示器。之后，在 PC 上透过串口运行 `getty` 等终端管理软件启动一个 Shell，就可以像使用任何一个熟悉的终端模拟器一样使用它。

为了正确地交换数据，终端与 PC 通信遵循不同的规范，它们规定了两端互相发送数据的格式。从 PC 到终端，有字符编码、光标控制、字符颜色、插入模式等信息；从 PC 到终端，每一个（或者一组）按键都对应着不同的一串字符串。我们完整地实现了被接受最为广泛的 `VT220` 标准[1]，它在 Linux、macOS 等操作系统中都有完善的支持。鉴于 `VT220` 的显示只是单色模式，为了更好的效果，我们还实现了 `xterm-256color`[2]模式下的大部分影响显示效果的指令，尤其是增加了对颜色的支持。我们实现了标准中的所有颜色模式与字符特效，遗憾的是受到实验板的限制，最终的输出会被化归为 512 色。

本项目可以与运行任何现代操作系统的 PC 配合，运行日常所用的命令行模式的软件，也可以用来连接网络设备等进行配置、调试工作。

本项目全部采用 `SystemVerilog` 进行编写。它带来了多个方便的新特性，如：

`logic` 类型 整合了 `wire` 和 `reg` 类型，自动推导几类寄存器模型，减少无谓的区分。

`always_comb`，`always_latch`，`always_ff` 块 可明确指定需要的逻辑类型（纯组合逻辑、锁存器、触发器），防止出现非预期电路和行为。

`struct` 类型 与 C 语言类似，可将一组值/信号进行单独赋值、统一绑定与传递。

`enum` 类型 与 C 语言类似，可以给常量赋名称，并有作用域限制，在状态机的设计中减少了出错的可能。

`typedef` 关键字 与 C 语言类似，可以给复杂的变量类型赋予别名，更直观好记。

`define` 与 `generate` 关键字 可以方便地在预处理阶段批量生成代码，以及在综合阶段批量生成元件，适合大量小元器件并行使用的场合。

[1] 介绍可见 `https://en.wikipedia.org/wiki/VT220`，以及下面给出的技术文档

[2] 介绍可见 `https://en.wikipedia.org/wiki/Xterm`

进一步地，SystemVerilog 在电路的测试验证中具有 VHDL 与 Verilog 不可比拟的灵活性、便捷性。我们强烈建议各位学习这门先进的硬件描述语言，也建议老师在今后的教学中考虑采纳它，与现代 VLSI 设计接轨。

另外，本项目还有以下的突出特性：

高分辨率 我们实现了 800*600@72Hz 的 VGA 信号输出，一屏幕能容纳 100 列 *50 行 =5000 个文字，相比标准要求的 80 列 *25 行 =2000 字提升了 250%。

高速度 通过对时序的精心优化，我们的解析与渲染模块都做到了全流水线运行，最高能运行在 120 MHz 的频率下。因此，我们能以 3M 的波特率与计算机串口进行通信，从而能够以 25FPS 流畅播放电影（由播放器渲染为动态彩色文字输出）。

高稳定性 我们充分发掘 SRAM 潜力，在渲染与显示器输出中采用了双缓冲机制，解决了屏幕撕裂、像素微小抖动等常见问题。

高模块化 我们在整个工程中，坚持高内聚低耦合的思想，将每个重要功能都抽取为独立的模块，减少代码冗余，也增强了可扩展性。同时 VGA、显存、键盘等模块通用性强，可用在今后的其他项目中。

1.3 项目分工

陈晟祺：

- 项目总体架构
- 键盘控制模块（PS2、译码）
- 显示控制模块（VGA、SRAM）

周聿浩：

- 终端命令解析与处理
- 项目综合调试
- 展示 Demo 设计

注：各模块的详细功能与设计可见下面各节，两人的工作量基本是均衡的。

2 架构设计

本项目中全面运用了模块化的设计思想。在最初设计时（见图2），我们为项目设计了六个功能模块（键盘译码、渲染控制、UART 发送与接收、VGA 信号发生与处理）和一个调试模块。然而，终端本身功能的特点，即其事实上不对用户指令进行任何除译码外的处理，决定了其实这个系统中存在两条独立的信号通路，即：键盘输入 → 按键译码 →UART 发送，UART 接收 → 指令处理 → 渲染 →VGA 输出。因此，在最终实现中（如图2），我们将项目分成了两个大模块，即键盘控制器 KeyboardController 与图形控制器 VideoController，二者分别处理上述的两条信号通路，完成各自的功能，互相独立。事实证明，这种解耦合的思想为我们的开发带来了很多的便利。

3 实现细节：键盘控制模块 KeyboardController

键盘控制器模块负责接收 PS/2 接口传来的数据，并将按键最终转换为 VT220 规范中指定的指令序列从串口发送给 PC。本模块主要由下列的子模块构成。

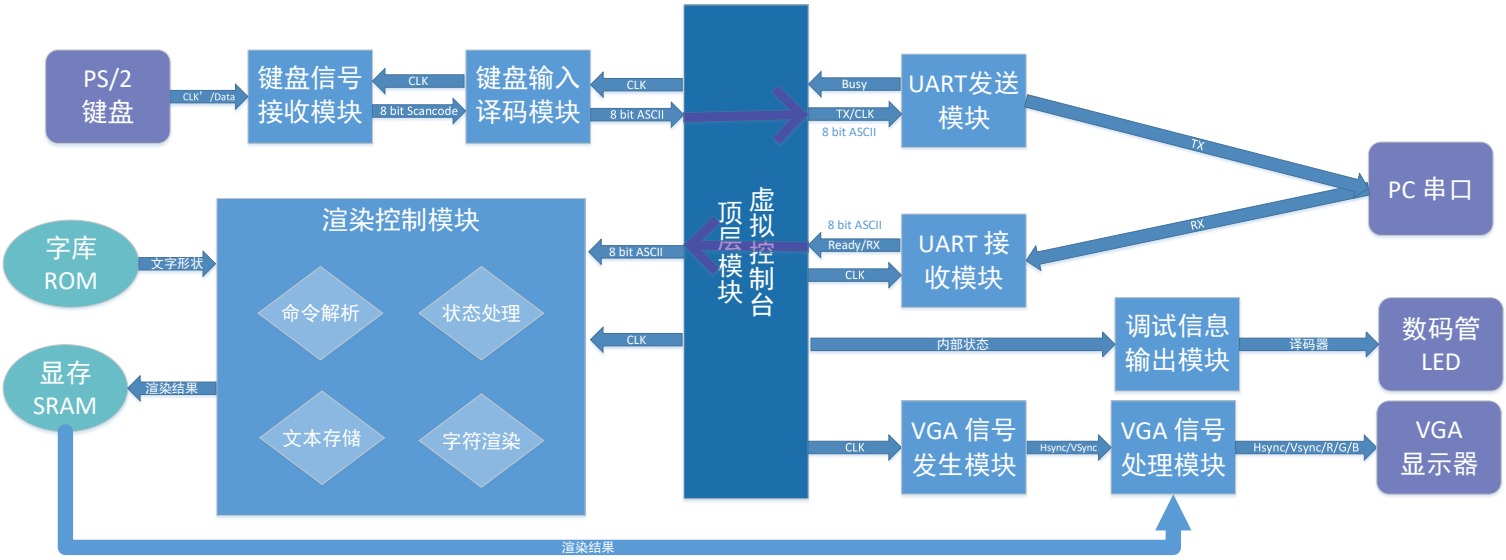


图 1: 项目设计架构

3.1 PS2 接收模块 Ps2Receiver

本模块采样由 PS/2 接口发送来的键盘时钟与数据信号，在每次正确收到一个扫描码（8 位二进制）后发出指示。由于键盘时钟远远慢于该模块运行的时钟频率，所以我们对键盘数据进行了 8 倍过采样，以最大程度地消除毛刺，保证数据的正确性。

3.2 扫描码译码模块 ScancodeDecoder

本模块用于将扫描码转换为控制命令序列。我们需要区分的情况有：

是否为特殊扫描码 特殊扫描码以 `E0` 为前缀，此时的扫描码有至少 2 字节长。其中比较有代表性的一部分可见表1所示。其中 `ESC` 表示 `0x1Bh`。较完整的内容可参见 `https://docs.microsoft.com/en-us/windows/console/console-virtual-terminal-sequences`。

表 1: 一些特殊按键扫描码与 Escape Sequence 对应关系

Key	Scancode	Escape Sequence
Insert	E0 70	ESC [2 ~
Delete	E0 71	ESC [3 ~
Left Arrow	E0 6B	ESC [D
Home	E0 6C	ESC [H
End	E0 69	ESC [F
Up Arrow	E0 75	ESC [A
Down Arrow	E0 72	ESC [B
Page Up	E0 7D	ESC [5 ~
Page Down	E0 7A	ESC [6 ~
Right Arrow	E0 74	ESC [C

非特殊扫描码的按键中也有部分会产生较长的 Escape Sequence，如 `F1` 到 `F10` 的功能键。为了节省篇幅，此处不再叙述。

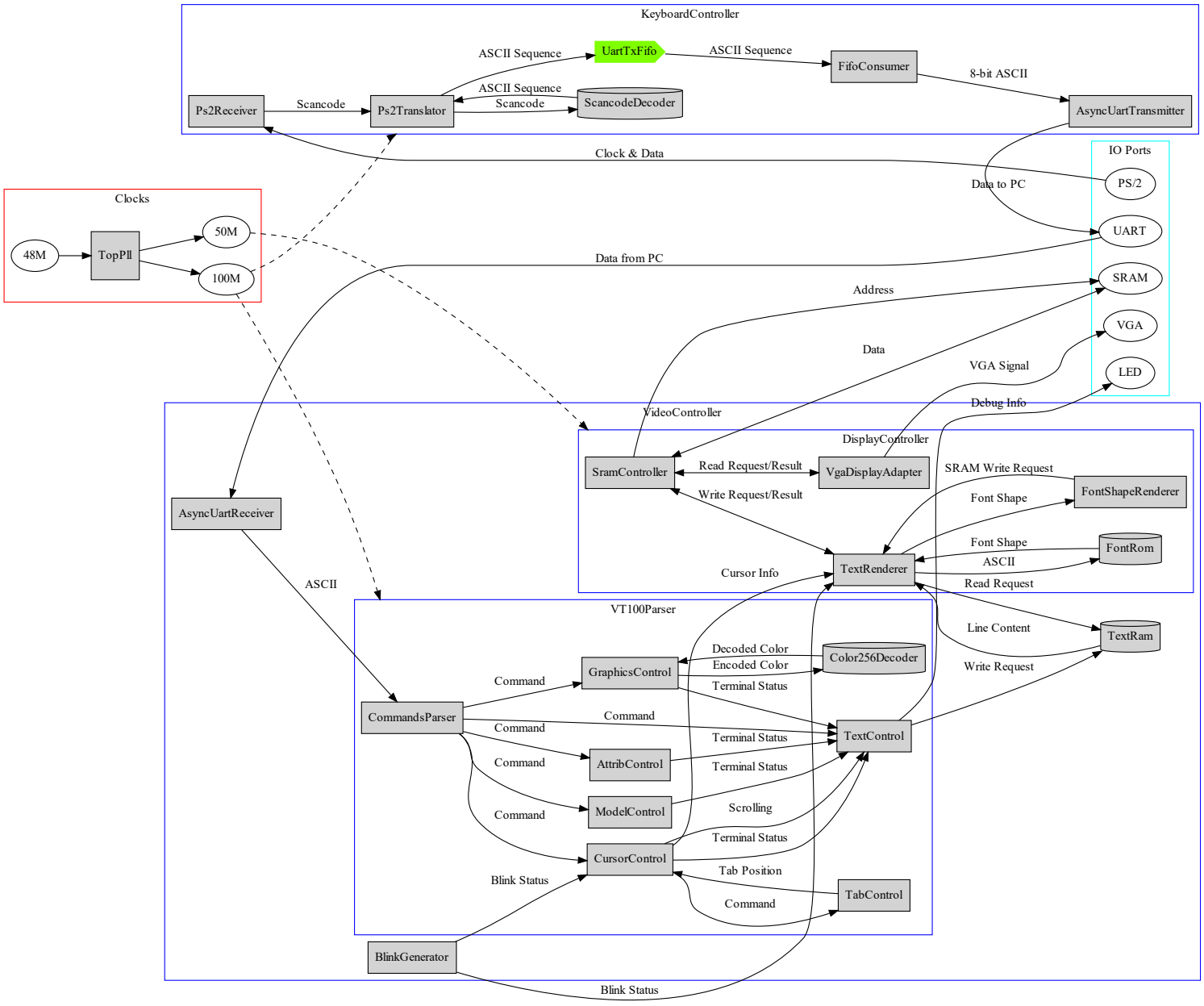

KeyboardController
ASCII Sequence
UartTxFifo
ASCII Sequence
FifoConsumer
8-bit ASCII
Ps2Receiver
Scancode
Ps2Translator
ASCII Sequence
Scancode
ScancodeDecoder
AsyncUartTransmitter
Clock & Data
IO Ports
PS/2
Data to PC
Clocks
48M
TopPll
50M
100M
UART
Data from PC
SRAM
Address
VGA
Data
VGA Signal
LED
Debug Info
VideoController
DisplayController
SramController
Read Request/Result
VgaDisplayAdapter
SRAM Write Request
Write Request/Result
FontShapeRenderer
Font Shape
Font Shape
FontRom
ASCII
AsyncUartReceiver
Cursor Info
TextRenderer
Read Request
ASCII
VT100Parser
Line Content
TextRam
Decoded Color
Color256Decoder
Encoded Color
GraphicsControl
Write Request
Terminal Status
Command
CommandsParser
Command
TextControl
Terminal Status
Command
AttribControl
Terminal Status
Command
ModelControl
Scrolling
Terminal Status
CursorControl
Tab Position
Blink Status
TabControl
Command
BlinkGenerator
Blink Status

图 2: 项目最终架构

是否按下 Shift 键 这部分的处理比较简单，这是由于 VT220 规范中不包含使用 Shift 的组合键，故只要正常处理 Shift 的功能（即输出大写字母/键帽上靠上的字符）即可。

是否按下 Ctrl 键 在 VT220 规范中，对于同时按下 Ctrl 与字母键发送的序列有详细的规定，可见 https://vt100.net/docs/vt100-ug/table3-5.html。需要特别处理的是 Ctrl 与特殊扫描码按键同时按下的情况，具体如表2所示。

表 2: Ctrl 与特殊扫描码按键对应的 Escape Sequence

Key	**Escape Sequence**
Ctrl + Up Arrow	ESC [1 ; 5 A
Ctrl + Down Arrow	ESC [1 ; 5 B
Ctrl + Right Arrow	ESC [1 ; 5 C
Ctrl + Left Arror	ESC [1 ; 5 D

3.3 PS2 转换模块 Ps2Translator

本模块负责接收扫描码，并将每一个键盘事件（如按下一个按键或一个组合键）对应的 Escape Sequence 送入 FIFO 中待发送。本模块由一个较复杂的状态机实现，如图3所示。

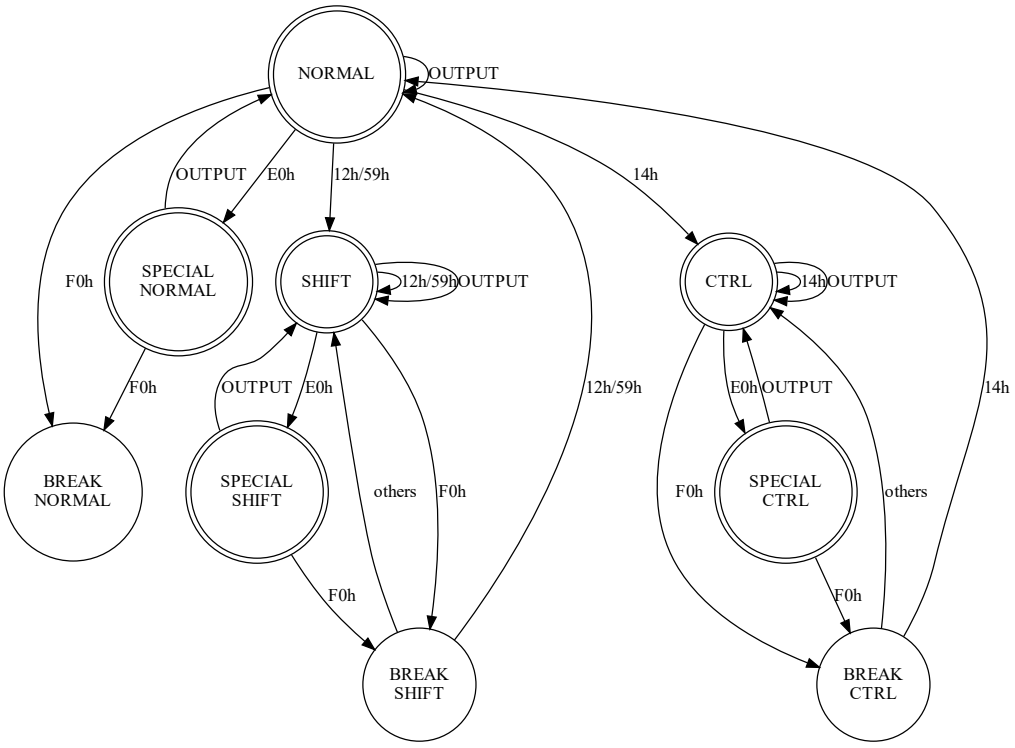


图 3: PS2 转换模块状态转移示意图

其中转移边即为收到的扫描码，OUTPUT 表示将此时收到的扫描码传给译码模块，并将得到的 Escape Sequence 推入 FIFO 中。所有双圆圈标记的状态为可能产生输出的状态。

此模块可以正确地处理重复按键、多个按键等常见情况。当得到的按键或组合没有在 VT220 规范中时，本模块不会产生任何输出。

3.4 FIFO 数据格式

PS2 转换模块与串口发送模块之间通过一个 FIFO 进行数据缓冲。由于每个 Escape Sequence 序列的长度不会超过 56 字节，FIFO 的位宽定为 64 比特，其中存储的数据格式如表3所示。

表 3: FIFO 中每条数据的格式

64	56	48	40	32	24	16	8	0
Length	Char 7	Char 6	Char 5	Char 4	Char 3	Char 2	Char 1	

3.5 FIFO 处理模块 FifoConsumer

此模块的功能比较比较简单，由于串口的发送比较慢，故使用该模块配合 FIFO 进行缓冲。此模块包含一个状态机，设计思路如下：

1. 等待，直到 FIFO 不为空
2. 从 FIFO 中读取一条数据（格式如表3），根据长度跳到第一个待发送字符之前
3. 读取一个字符，向串口送出，等待直到串口发送完成
4. 如果还有字符没有发送完成，跳到 3，否则跳到 1

事实上，由于要发送的数据总长度较短，我们直接使用状态名而不是一个变量来记录当前剩余的字符数量，根据 Length 属性跳转到相应状态。这样实现的好处是节省了不必要的时钟周期，减少了逻辑单元的复杂度，但是导致状态数量增长到 31 个。由于这些状态都可分为几个大类（读取、等待、跳转），代码比较类似，借助于 SystemVerilog 的 define 关键词与宏展开，实现一个这样一个看似复杂的状态机是很简洁的。

4 实现细节：图形控制模块 VideoController

图形控制器的主要功能是接收并解析 PC 发送的指令数据，对屏幕显示的内容进行相应的更改。其中又分为五个模块，其中 AsyncUartReceiver 处理串口的接收，VT100Parser 解析这些信息，并将当前屏幕上应该显示的内容存储到片内 RAM 例化成的 TextRam 部件中。DisplayController 从 RAM 中读取文本信息，转化为图像数据存储在显存中，并转换为 VGA 信号输出。还有一个 BlinkGenerator 模块用来产生一个统一的光标闪烁信号供各模块使用。各个模块的详细说明见下面各节。

4.1 RAM 存储格式

为了让 VT100Parser 的输出能够正确被 DisplayController 识别并渲染，我们设计了一套存储格式。RAM 的深度是 50，对应终端的 50 行；位宽是 $100 \times 32\text{bit} = 3200\text{bit}$，即每一个字符 32 比特；每条数据中靠前的字符在较低位。具体的数据格式定义可见表4。

表 4: RAM 中每个字符的存储格式

31	30	29	28	19 ... 10	8 ... 0
K	N	B	U	BG Color	FG Color	CS	ASCII

表 5: 终端内部状态表

`cursor.x`	光标的 X 坐标
`cursor.y`	光标的 Y 坐标
`graphics.bg`	背景色
`graphics.fg`	前景色
`graphics.effect`	表4中所列特效
`mode.origin_mode`	Origin 模式
`mode.auto_wrap`	AutoWrap 模式
`mode.insert_mode`	Insert 模式
`mode.line_feed`	LineFeed 模式
`mode.cursor_blinking`	光标是否闪烁
`mode.cursor_visibility`	光标是否可见
`attrib.scroll_top`	滚动区域顶
`attrib.scroll_bottom`	滚动区域底
`attrib.charset`	字符集
`prev_data`	上一个输入的字符

其中 K、N、B、U 是字符特效标记，分别表示闪烁、反色、变亮、下划线；`BG Color` 与 `FG Color` 分别是当前字符块的背景和前景色，和 VGA 信号的格式一致，依次为 3 比特的 RGB 值；`CS` 表示当前字符集（Charset），这是由于 `VT220` 规范支持切换字符集，使用特殊字符进行表格绘制等操作，我们目前没有使用这个字段；最后的 `ASCII` 字段存储了当前字符的编码。

4.2 终端命令解析与处理 `VT100Parser`

对于终端的命令，我们支持 VT220 标准[3]下几乎所有的会影响显示结果的命令，以及 XTerm[4]下常用的光标控制以及字符编辑命令，可以支持解析绝大多数程序的输出。目前所支持的命令见表6。

终端具有多种内部状态，例如光标位置、光标是否闪烁和滚动区域等，这些状态均会影响到各个命令的行为。我们的命令处理模块主要维护了表5中所列的内部状态。各类状态的具体维护由各个子模块进行处理。

4.2.1 命令解析模块 `CommandsParser`

终端的命令主要的格式是

Introducer Parameters Terminator

[3] `https://vt100.net/docs/vt220-rm/`

[4] `http://invisible-island.net/xterm/ctlseqs/ctlseqs.html`

它们表示命令开始、命令参数和命令结束。其中 *Introducer* 主要是由 ESC 或 CSI[5]构成，*Parameters* 可以有多个，并且用分号分隔。

命令解析模块主要是将串口收到的字节流解析为对应的命令，并且将参数提供给后续模块的处理。

命令的解析是通过如图5所示的状态机来进行实现，主要在 `CommandsParser` 模块中。在解析完成一条命令后会产生一个 `commandsReady` 信号，并且附带额外的命令参数等信息以供后续模块执行对应操作。

大部分命令的参数个数以及格式均是固定的，解析较为简单。有两类特殊的命令，其参数个数不定，它们分别是控制显示字符的颜色等信息的命令，以及设置/清除终端某些状态的命令。这两类命令的参数均为零或多个由分号分隔的数字构成，参数的个数任意。

对于这两个命令的解析，我们采用另外的方法：

1. 当一条命令开始时产生一个 `commandsReady` 信号并且对应 INIT_PNS 命令，后续需要处理这两类命令的模块在收到此命令时重置其中状态机。

2. 每当获取到一个参数后产生一个 `commandsReady` 信号，并且对应一个 EMIT_PNS 的命令，后续需要处理这两类命令的模块在收到此命令后根据所得到的参数在其自身内部的状态机内进行处理（具体的处理方法见各模块的描述）。

3. 当真正解析到该条命令结尾时，产生对应命令的信号，后续模块将之前所记录的信息应用到终端的状态中更新其状态。

表 6: 当前支持的命令

命令名称	命令格式	命令行为
CUU	`ESC [ Pn`[6] `A`	光标上移
CUD	`ESC [ Pn B`	光标下移
CUF	`ESC [ Pn C`	光标前进
CUB	`ESC [ Pn D`	光标后退
CNL	`ESC [ Pn E`	光标下移到行首
CPL	`ESC [ Pn F`	光标上移到行首
CHA	`ESC [ Pn G`	光标水平移动
VPA	`ESC [ Pn d`	光标垂直移动
CUP/HVP	`ESC [ Pn; Pn f/H`	设置光标位置
REP	`ESC [ Pn b`	重复上一个输入的字符
DL	`ESC [ Pn M`	删除行
IL	`ESC [ Pn L`	插入行
ED	`ESC [ Pn J`	删除部分显示的文本
EL	`ESC [ Pn K`	删除部分当前行文本
ECH	`ESC [ Pn X`	清除部分字符
DCH	`ESC [ Pn P`	删除部分字符
ICH	`ESC [ Pn @`	插入空白字符

[5]即 ESC [

[6]Pn 表示数值参数，可以省略

HTS	`ESC H`	设置 TabStop
TBC	`ESC [ Pn g`	清除 TabStop
SETDEC	`ESC [? Ps`[7] `h`	设置 DEC 模式
RESETDEC	`ESC [? Ps l`	清除 DEC 模式
SETMODE	`ESC [ Ps h`	设置模式
RESETMODE	`ESC [ Ps l`	清除模式
NEL	`ESC E`	新行
RI	`ESC M`	回到上一行
IND	`ESC D`	转到下一行
DECSC	`ESC 7`	保存光标状态
DECRC	`ESC 8`	恢复光标状态
SU	`ESC [ Pn S`	向上滚动
SD	`ESC [ Pn T`	向下滚动
DECSTBM	`ESC [ Pn; Pn r`	设置滚动区域
SS2	`ESC N`	设置字符集
SS3	`ESC O`	设置字符集
SCS0	`ESC ( A/B/0/1/2`	设置字符集
SCS1	`ESC ) A/B/0/1/2`	设置字符集
SGR	`ESC [ Ps m`	设置颜色等字符信息

本模块中还初始化了下列的子模块。

4.2.2 光标控制模块 `CursorControl`

该模块主要负责控制光标的位置。光标的位置受到多种命令和终端状态的控制：

- 当新输入一个字符时，光标右移一格。当光标处于行末时，若处于 AutoWrap 模式，则跳到下一行的第一个位置，否则不动。
- 当输入 LF、FF 和 VT 时，光标移动到下一行的同一列。若处于 LineFeed 模式则还会移动到行首。
- 当输入 BS 时，光标回退一格，若处于行首则不动。
- 当输入 HT 时，移动到下一个 TabStop 位置。
- 当滚动区域改变时，光标移动到原点。
- 对于 REP 命令，光标的行为和插入字符相同。
- 对于 IL、DL 命令，光标移动到行首。
- 其余直接控制光标的命令，诸如 CUU、CUD、CUF 和 CUB 等。

[7] Ps 表示由分号分隔的个数不定的数值参数列

光标的位置的原点取决于是否处于 Origin 模式。若处于 Origin 模式，则原点位于滚动区域的原点，否则位于整个屏幕的原点。

对于部分命令,其对光标的操作会触发滚动屏幕的操作。这会引起一个名为 `scrollReady` 的信号被设置，并且附带一部分滚动屏幕的参数，交由 `TextControl` 模块处理。

对于制表符的处理，当输入这类字符时，该模块不会直接处理，`TabControl` 发现该命令会寻找下一个 TabStop 的位置，并且在找到后产生一个 `TabReady` 信号并且附带查找到的位置。本模块在接受到这样的信号时再将光标设置到对应位置。

4.2.3 制表符控制模块 TabControl

该模块主要负责 TabStop 的设置和光标位置的寻找。默认情况下，每 8 个字符会有一个 TabStop。

当输入一个 VT 字符后，会引起光标跳转到下一个 TabStop 的位置。该模块会通过一个状态机进行循环，查找下一个 TabStop 位置，并且反馈给 `CursorControl` 模块设置光标位置。

此外，有 TBC 和 HTS 两条命令可以设置不同的 TabStop 位置，也由该模块进行处理。

4.2.4 字符控制模块 TextControl

该模块负责 TextRam 的读写，用于施加编辑命令产生的改变。控制整个模块功能的状态机如图4所示。

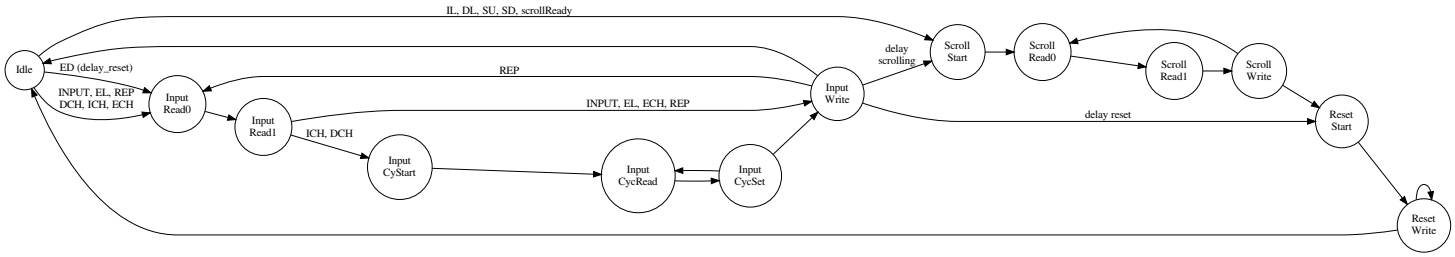


图 4: 字符控制模块状态图。

该模块对 TextRAM 的操作主要由三种基本操作组成：

行内编辑 主要负责对一行内的信息进行编辑，一共分为三种编辑方法，

1. 将指定行某个区间的所有字符设置成特定字符（记为 A 类命令）。
2. 将指定区间的字符删除，右侧字符左移并且在空白处填充空字符（记为 B 类命令）。
3. 在指定区间插入空字符，原有字符右移，若超出屏幕则丢弃（记为 B 类命令）。

其状态主要为 InputRead0、InputRead1、InputCycStart、InputCycRead、InputCycSet、InputWrite 组成，它们的意义分别为

- *InputRead0.* 发送读请求，读取所需要编辑的行。

- *InputRead1.* 等待数据到达。如果是 A 类命令，则直接用一个组合逻辑进行计算输出值，并且跳转到 InputWrite；如果是 B 类命令，由于组合逻辑过于复杂，跳转到 InputCycStart 通过时序逻辑来完成。
- *InputCycStart.* 将当前行的数据锁存，并且初始化后续需要的变量（当前所编辑的列，当前所需要读取的列）。
- *InputCycRead.* 从锁存的行数据中读取当前所需要读取的列对应的字符，并存储。
- *InputCycWrite.* 将存储的字符写入当前所编辑的列。并且更新当前列和所需要读取的列。
- *InputWrite.* 将编辑完成的行写入。

屏幕滚动 主要负责将某区域上移/下移指定行数。

该部分主要由 ScrollStart、ScrollRead0、ScrollRead1 和 ScrollWrite 组成，它们的意义分别为

- *ScrollStart.* 设置起始的行。
- *ScrollRead0.* 将读请求发送给 TextRam。
- *ScrollRead1.* 等待数据到达。
- *ScrollWrite.* 将数据写入对应行。

多行清空 主要负责将某区域填充为空字符。

该部分主要由 ResetStart 和 ResetWrite 两个状态控制。其中 ResetStart 负责设置起始的行，而后 ResetWrite 不断往下，每次向 RAM 中写入一个空行直到所需要清除的区域结束为止回到 Idle 状态。

该模块所处理的所有命令均可以通过这三种基本操作的组合构成，组合的操作基本有

滚屏 在“屏幕滚动”结束后跳转到“多行清空”，将因为滚动产生的空位设置为空字符。

插入编辑 当光标位于行末且处于插入编辑模式，在输入一个字符后会输入一个字符，并且产生一个新行。此种类型优先进行“行内编辑”，并且设置一个 `delay_scrolling` 标记，在行内编辑结束后跳转到“滚屏”。

混合删除 对于 ED 命令，会删除光标至屏幕末尾所有的字符，这分为两个步骤。首先是“行内编辑”清除光标所在行的字符，并且设置 `delay_reset` 标记，在行内编辑结束后跳转到“多行清空”再清除剩余的内容。

跨行插入 对于 REP 命令，其功能要求是插入指定字符，这可以跨越多行。这类操作会使用“行内编辑”，但同时设置一个 `rep_mode` 标记，同时设置必要的插入到何处的参数。之后“行内编辑”一行结束后检查是否到跨行插入的最后一行，如果不是则继续跳转到“行内编辑”初始处。

对于不同的命令，分别初始化不同的组合操作参数进行处理。

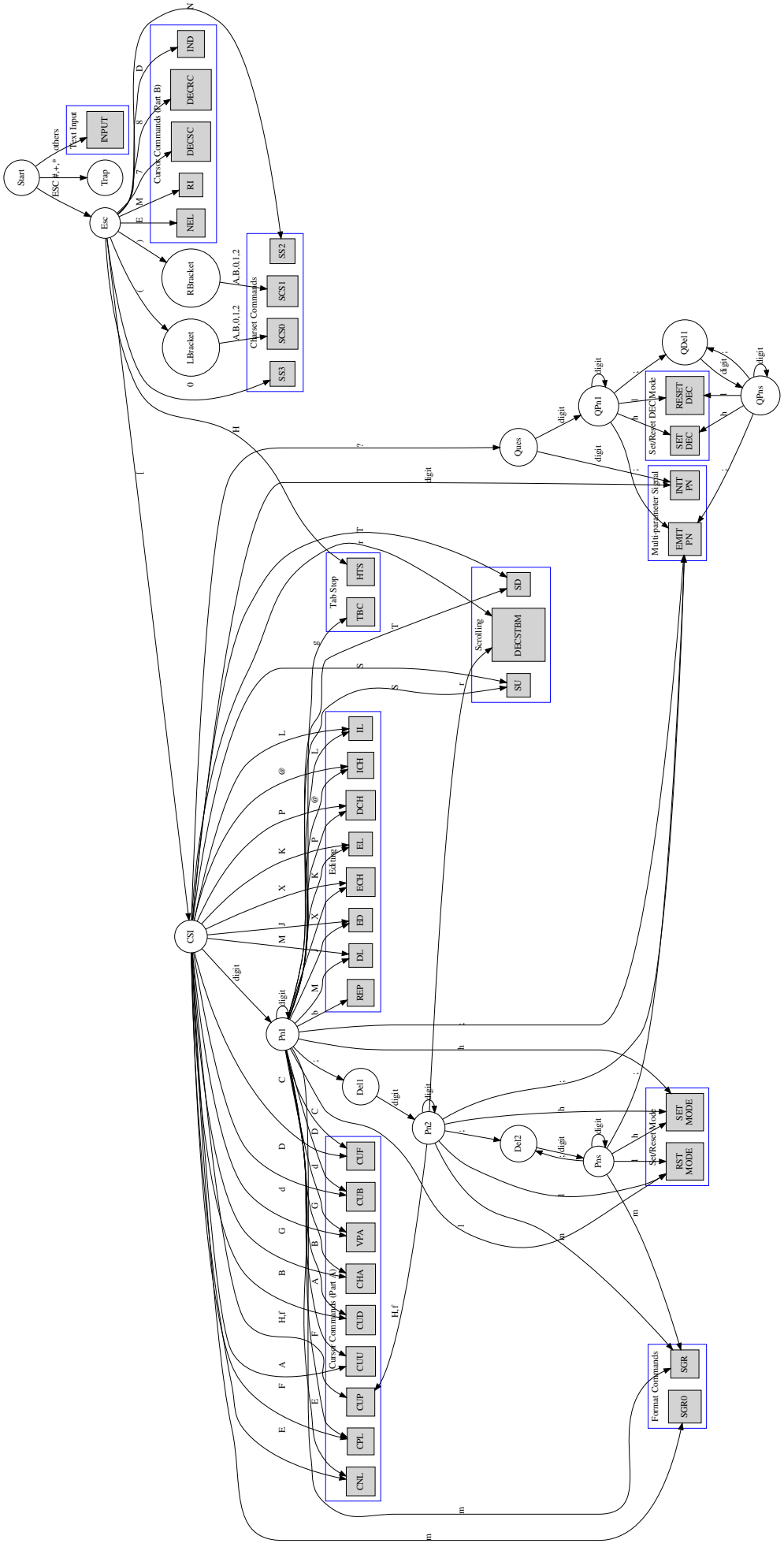


图 5：命令解析状态图，圆形节点为状态，方形为命令。

4.2.5 图形控制模块 `GraphicsControl`

该模块主要负责处理字符的颜色和特效（下划线、闪烁、反色和增亮）。对应的命令为 SGR，其参数个数不定，且共分为 3 种类别

1. 仅用一个参数表示的属性，这部分对应于 VT220 的标准。
2. 需要三个参数表示的属性，这是在一个 256 色的颜色表中指定对应颜色，其格式为 *48/38; 5; Pn*，其中 38 表示前景色，48 表示背景色，Pn 表示颜色的编号。
3. 需要五个参数表示的属性，这是用 RGB 指定颜色，其格式为 *48/38; 2; R; G; B*，其中 38 表示前景色，48 表示背景色。

这条命令可以在一个命令内同时设置多个属性，例如 *ESC [38; 5; 100; 48; 2; 0; 0; 255m* 就是设置前景色为编号为 100 的颜色并且设置背景为蓝色。

对应的状态机见图6，具体操作见 “命令解析” 中多参数命令的处理方法。

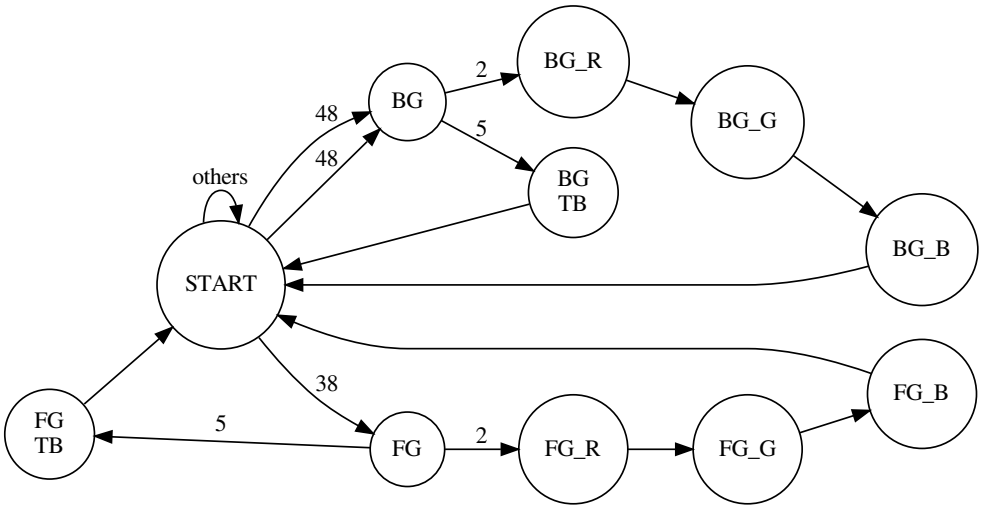


图 6: 图形控制模块状态图

4.2.6 属性控制模块 `AttribControl`

该模块是对滚动区域、字符集等属性设置命令的处理，较为简单，仅仅在得到命令后直接更新即可。

4.2.7 模式控制模块 `ModeControl`

该模块和图形控制模块同样是一个对应于多参数命令的模块，但是各个属性均可只用一个参数表示，在记录完毕所需要更新的模式后，在更新命令到来时直接更新即可。

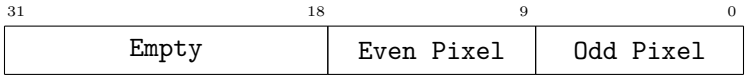
4.3 显示控制器 DisplayController

显示控制器的主要功能由三个模块实现：`SramController` 负责控制 SRAM 的读写，`TextRenderer` 负责读取 RAM 中存储的文本，渲染为像素数据并存储在 SRAM 的显存中；`VgaDisplayAdapter` 负责产生 VGA 信号，从 SRAM 显存中读取像素数据并输出。各部分的详细实现如下。

4.3.1 SRAM 控制器 SramController

我们使用片外的 SRAM 作为显存，并且采用了双缓冲机制，即使用两块显存 A 与 B，当渲染器向 A 写入时，显示模块从 B 进行读取，当二者均完成后进行交换，如此循环。这样保证了渲染和输出过程都有一致性，不会产生画面撕裂、闪烁等异常现象。由于使用的 SRAM 读写速度均在 20ns 左右，因此 SRAM 控制器最高时钟频率为 50MHz，这就导致了读/写实际的时钟频率只有 25MHz。而由于我们采用的 VGA 时序要求的像素频率为 50MHz，因此我们必须至少在一次读写中操作相邻的两个像素，才能保证输出图像是正常的。因此，我们设计了表7中所示的 SRAM 存储结构。对于屏幕上同一行上相邻的 (x, y) 与 $(x, y+1)$ 两个像素，它们相对于当前显存基址的偏移都是 $\frac{x+y}{2}$。

表 7: SRAM 中每条数据的存储结构



由于 VGA 模块的读取请求和渲染器的写入请求并非总是间隔进行（如 VGA 消隐区时不进行读取，渲染器读取文本数据时不进行写入），因此简单的 Round-Robin 算法不能满足我们的需要。而又因为渲染模块是由状态机驱动，可以任意进行等待；VGA 模块的读取请求如果失败，就会出现渲染问题。为此，我们为 SRAM 控制器设计了优先机制，即其在每次时钟沿时优先处理 VGA 模块的读取请求，如果没有，再处理渲染模块的写入请求。测试证明，这样的机制能够充分利用每一个时钟周期，没有浪费，也保证了渲染与显示模块的正常工作。

需要特别注意的是，正如往年同学在实验报告中指出的，我们原本使用的实验板的 SRAM 请求都经过了控制 FPGA 的转发，读写延时都比预期要长。我们实测，这样会导致控制器无法运行在 50MHz 的频率下。为了减少不必要的麻烦（如重写烧写控制 FPGA，或者在 SRAM 一条记录中存储更多像素），我们更换了另一块 FPGA 直接连接到 SRAM 的实验板，避免了这一问题的发生。

4.3.2 文本渲染模块 TextRenderer

文本渲染模块的实现是一个状态机，工作如下：

1. 从 RAM 中读取下一行文本（可循环）
2. 如果未到行末，读取下一个字符，并从 `FontRom` 中读取相应字形；否则跳回 1
3. 将字形、属性（如颜色、特效等）和当前字符第一个像素在显存中的地址传递给内部的子模块 `FontShapeRenderer`，指示开始渲染
4. `FontShapeRenderer` 渲染完成后，跳回到 2

其中 `FontRom` 是一个存储了 ASCII 中 256 个字符对应的形状的 ROM，每个字符宽 8 像素，高 12 像素，地址即为自身的 ASCII 编码。`FontShapeRenderer` 是用于渲染一个字的子模块，也是一个状态机，工作如下：

1. 等待父渲染器的开始信号，并接受传递的信息
2. 如果未到最后一个像素，计算下一个像素的颜色并锁存数据；否则报告完成并回到 1
3. 计算下一个像素的颜色，计算这两个像素在显存中的具体地址，时钟沿末向 SRAM 控制器发出写请求
4. 如果控制器报告写请求完成，跳到 2，否则等待

4.3.3 VGA 显示模块 `VgaDisplayAdapter`

VGA 控制器负责产生 VGA 图形信号（我们使用的实验板上实际是 HDMI 接口，不过我们需要给出的信号格式是一致的）。为了使我们的终端实用性更强，我们实现了 800*600 输出分辨率，也即 50 行 *100 列，每一个单元格一个字符。为了配合现有的 SRAM 等模块工作，结合 `http://tinyvga.com/vga-timing` 给出的 VGA 时序规范，我们选取了 72Hz 刷新率，使得输出的像素频率刚好为 50MHz，具体的时序可见表8。

表 8: 800*600@72Hz 的 VGA 信号时序规范

Scanline part	**Pixels**	**Time (μs)**	**Frame part**	**Lines**	**Time (ms)**
Visible area	800	16	Visible area	600	12.48
Front porch	56	1.12	Front porch	37	0.7696
Sync pulse	120	2.4	Sync pulse	6	0.1248
Back porch	64	1.28	Back porch	23	0.4784
Whole line	1040	20.8	Whole frame	666	13.8528

由于需要预读取像素数据，VGA 控制器也被设计为一个状态机。VGA 的时序信号发生模块是独立于状态机工作的，保证信号严格遵守规范输出。状态机的工作如下：

1. 重置时序发生模块，时钟沿末发送对最早两个像素的读取请求
2. 锁存得到的两个像素数据，时钟沿末输出奇数像素
3. 时钟沿末输出偶数像素，如果下一个像素依旧在图像区域，则计算出需要读取的内存地址并在时钟沿末发送读取请求，跳转到 2；否则跳转到 4
4. 不断等待，直到下一个像素回到图像区域，计算出需要读取的内存地址并发送读取请求，跳转到 2

5 其他说明

本项目是基于 Cyclone IV EP4CE55F23I7 开发的，具体的运行说明请见源码根目录下的 `README.md`。

感谢计研 173 张宇翔学长提供的项目灵感和实验板，他在我们的开发过程中帮助我们解答了各类疑难问题，也协助我们进行了一些调试工作。同时也感谢老师和助教的大力支持。缺少了诸位的帮助，写好这个项目是不可能的。